Mesocode: Optimizations for Improving Fetch Bandwidth of Future Itanium® Processors

Marsha Eng[1], Hong Wang[1], Perry Wang[1], Alex Ramirez[2], Jim Fung[1], John Shen[1]

[1]Microprocessor Research Labs, Intel Labs
Santa Clara, CA 95054

[2]Computer Architecture Department
UPC, Barcelona, Spain

ABSTRACT

In this work, we introduce the concept of mesocode, a code format exhibiting the benefits of both traditional macro-code and microcode. Like macro-code (i.e. instruction set architecture), mesocode format is architecturally visible to post-pass software tool. In addition, like micro-code, mesocode is implementation specific and legacy free. In this paper, we focus on a mesocode format designed specifically to improve fetch bandwidth efficiency of future Itanium® processors. The mesocode instruction format is more compact than the canonical Itanium® ISA. This mesocode design is based on *stream*, a sequence of dynamic instructions between two consecutive *taken* branches. The machine supporting the stream-based mesocode can predict, fetch, and execute these streams efficiently. Our studies show that for a set of SPEC2000 benchmarks less than 10% of the streams cover over 90% of dynamic instructions, indicating that optimizing a small set of streams can potentially improve performance for a large portion of the program execution. For dynamic (a.k.a. out-of-order) processor implementations, the front-end fetch bandwidth is critical to performance. Our experiments show that average performance improvement of 32% can be achieved through mesocode optimization, which both improves code density of the frequently executed portions of the code and enables more timely and effective instruction prefetches.

1. INTRODUCTION

This paper presents a case study regarding how to apply complexity-effective design principle to tackle a particular challenge for implementing future out-of-order Itanium® processor, namely, the complexity posed by the instruction encoding format in the Itanium® instruction set (ISA).

The Itanium® ISA, like traditional VLIW, suits wide in-order EPIC styled implementation schemes. For example, multiple instructions are bundled together to form the basic fetch unit, in which a template field is explicitly encoded to provide ancillary information about each instruction inside the bundle. In addition, each instruction is also encoded with bit-field for the predicate register due to use of predication in the ISA. Should compiler fail to find an independent instruction for a bundle, a NOP is usually used to fill in to ensure the bundle conform to one particular template encoding. Obviously, NOP, template, predicated false instruction and predicate register bits can contribute to waste of processor resources such as cache footprint, fetch bandwidth, instruction queue etc. Consequently, the wastage can result in performance loss.

We introduce the concept of mesocode, a code format designed to improve fetch bandwidth efficiency of future Itanium® processors. Mesocode introduces modest changes at the ISA level in an implementation specific and legacy-free format that is unobtrusive to the original code. For the specific implementation studied in this paper, mesocode is used to express the frequently execution portions of the original code in a more compact and efficient manner and the optimized mesocode is attached as appendix to the original code. On processor supporting mesocode, at run time, mesocode can be frequently executed in place of the counterparts in the original code to enable more efficient use of the machine resources. On processor that does not support mesocode, however, the mesocode will simply not be activated and the original code will run canonically thus ensuring backward and forward compatibility.

Our study bases mesocode on streams, which are sequences of dynamic instructions between two consecutive taken branches [24]. Our evaluation shows that about 10% of the unique streams cover over 90% of the dynamic instructions, making them good candidates for mesocode encoding. Optimizations made to a small number of streams effectively benefit the entire program.

The key contribution of this paper is to demonstrate that mesocode can improve fetch bandwidth by reducing wasteful instructions brought into the pipeline. In addition to benefiting from effective prefetch of streams, the removal of NOPs from streams results in a significant gain, about 32% on an out-of-order machine. The improvement in code density stemming from NOP removal not only increases cache utilization but also reduces resource contention in the front-end pipeline for resources such as the instruction queue.

The rest of the paper is organized as follows. In Section 2, we highlight how current architectures facilitate a loss of fetch bandwidth. Section 3 introduces the details of mesocode. Section 4 describes the performance evaluation methodology including simulation infrastructure and workload setup. Section 5 presents evaluations of a range of mesocode optimizations and their performance impact. Section 6 reviews related work. Section 7 concludes the paper.

2. INSTRUCTION ENCODING IN THE ITANIUM® ARCHITECTURE

EPIC ISAs, including the Itanium® ISA, uses template-carrying bundle as atomic unit of fetch and execution. The template actually makes it possible to decipher the types of instructions in

the bundle well before the instructions are decoded. Individual instructions inside a bundle act more like micro-ops and will be referred to as such to avoid confusion. Stop bits are used to express parallelism (for instructions between stop-bits) and data dependency (for instructions across stop-bits) behavior. The Itanium® ISA also includes predication and static branch hints on a micro-op level, which in conjunction with the stop bits and templates, could be used to express program behavior at granularity beyond the traditional basic block level.

The problem with forcing micro-ops into fixed issue templates is that NOPs are introduced to the code when no useable instructions can be found to fill out the rest of the template. These NOPs dilute the code density and degrade cache and pipeline utilization by taking up valuable space and pipeline resources that could be filled with useful instructions. The effective fetch bandwidth is reduced due to the pollution from these wasteful instructions. Predication can have the same effect in that instructions that are predicated false at runtime effectively become NOPs in the dynamic code stream, which occupy resources and degrade the IPC. Another problem with using fixed issue templates is that branch target is required to be bundle aligned. This can introduce cache line fragmentation when the cache line is bigger than a bundle. When a taken branch or a branch target is not aligned to the cacheline, then the rest of the cache line will be waste, which reduces effective usage of the fetch bandwidth. These problems can be solved with a flexible instruction encoding that can avoid emplicit encoding of NOPs and prevent misalignment to the cache line due to branches.

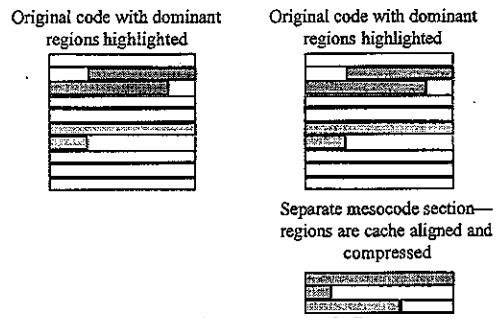

Figure 1. Static Mesocode Layout

3. MESOCODE

We introduce the concept of mesocode as one means of recharacterizing the code stream in a more efficient manner that addresses the problems of code density, instruction misspeculation, and lack of information about aggregates of micro-ops. Mesocode can be viewed as an intermediate form of binary code representation between macrocode (e.g. the original Itanium® ISA) and the machine-specific internal code (instruction syllables or micro-ops). The idea is to introduce modest enhancement to the macrocode ISA to express the underlying micro-ops in a legacy-free and less constrained format.

To achieve this in the Itanium® ISA, we use two previously unused templates to demarcate the start and end of mesocoded regions of microcode, which consist of instruction syllables or micro-ops that are not restricted by any other Itanium® templates. The microcoded regions are kept separate as appendices to the original code and are unobtrusive to the original program, as illustrated in Figure 1. In our implementation, the microcode redundantly expresses frequently executed portions of the original code, although they are potentially encoded very differently. As far as the original code is concerned, the mesocode is no different from, say, instructions for a co-processor.

In our version of mesocode, the code regions are executed speculatively. At the start of the mesocode section is a table mapping the original addresses of the starting instructions to their new realigned instruction addresses in the mesocode section. A monitor at speculatively redirects the instruction pointer to the realigned address on its next invocation. If any instruction in the mesocoded region causes an exception or executes incorrectly, program execution can always fall back to the original code. As a result, the mesocode-enhanced binary is guaranteed to be correct because it does not modify the original code; in the worst case, the original code is executed. Separating the mesocoded sections from the original code also ensures that the binary is backwards compatible.

In addition to the ISA modifications, mesocode requires modest hardware changes to be effective. Mesocoded instructions are the same as Itanium® ISA instruction syllables, so the portions of the core devoted to executing instructions and the caches and memory subsystems remain the same. The two templates marking the start and end of the microcode regions are now used as "escapes" from the original code stream. When the decoder encounters the start template, it switches to a special mesocode decoder that can handle the micro-ops. When the decoder encounters the end template, it switches back to the standard decode mechanism. The fetch mechanism is changed to recognize the new escape templates and fetch continuously until it reaches the end of the mesocoded region. The instruction issue for mesocode regions does not have to check for the template because it is nonexistent in mesocode encoding Within the regions, microcode should be scheduled in such a way that the instruction issue does not have to check for data dependencies and can simply issue the instructions. Since this instruction issue is different from that for the original Itanium® ISA, they essentially work in parallel with each other. That is, mesocode and original code can coexist with effectively little or no impact on each other. Finally, there is a small amount of instruction monitor and redirection logic to speculatively redirect the instruction pointer to the mesocode upon seeing trigger points from the original code being retired, and back to the original code when it reaches the end of the mesocode.

There are many ways to find instructions for the mesocoded regions. Our implementation uses a trace-driven post-pass compiler to identify frequently executed streams, or contiguous instructions executed in between two taken branches. The resulting mesocode is stream-based. Such an offline mechanism allows for easy morphing of the mesocoded regions—for example, it is possible to remove wasteful instructions and insert prefetching hints at specific locations in each region. An example of a runtime mechanism that is not studied here could closely resemble that used to find traces for the trace cache [16]

but uses the existing memory subsystem, thereby reducing the hardware footprint over the trace cache.

Because mesocoded microcode are treated differently from the rest of the code stream, their encoding is effectively unconstrained and does not have to match the original code. Thus there is flexibility within the microcode to reschedule and predecode portions of the instructions, add in such hints as prefetching directives, and realign the regions to better suit the instruction fetch mechanism. The different encoding means that microcode does not need the extra template bits used in the Itanium® ISA, which further means a potentially more compact encoding and more efficient use of bits. With additional information about the dynamic code stream and issue capabilities, we could further compress the mesocode by eliminating wasteful instructions and realigning the instructions.

3.1. Improving the Code Density

As already discussed, removing the template restrictions from the macrocode ISA when defining the mesocode means there are no requirements on how the micro-ops need to be ordered, except for producing correct output. Without the templates, it is possible to address the wasteful instructions that would have appeared in the original macrocode and microcode code streams. For this paper we study three major classes of wasteful instructions that degrade code density. Misaligned instructions can be found in most architectures, and NOP and predicated false instructions are of particular interest to EPIC architectures such as the Itanium® architecture.

3.1.1. Misaligned Instructions

Instructions may be misaligned in one of two ways. First, a taken branch may appear in the middle of a cache line, meaning the control flow will jump out of a cache line when it reaches the end of the basic block. Wasteful instructions in this case appear at the end of the cache line. Second, the target of a taken branch may appear in the middle of a cache line. The control flow will ignore the instructions at the start of the cache line. Wasteful instructions appear at the start of the cache line in this case. On the Itanium® architecture, the rule for instruction bundling defines a unit bundle as a sequence of three instructions and restricts that a branch target has to be bundle aligned. However if the cache line in specific processor organization is larger than a bundle, a target bundle can fall in the middle of the cache line. Fetching this target will incur misalignment problem since the prior bundle in the same cache line will be dropped eventually, though only after potentially expensive occupancy of pipeline resources.

Given the free-form encoding allowed by mesocode, it is possible to align new microcode regions to the start of the cache line, eliminating the fragmentation due to branch targets. This simple method does not eliminate the fragmentation due to a taken branch at the end of the microcode region; however, in eliminating the fragmentation at the start of the microcode, the total amount of misalignment is reduced. This fragmentation at the end of the cache line is less performance critical since program flow is expected to leave at this point.

3.1.2. NOPs and Predicated False Instructions

Instruction bundles and templates force instructions to be represented in a limited number of combinations to ensure they can be issued in a predefined specific order. This restriction was introduced to Itanium® architecture due to historic reason related to anticipated VLIW implementation of Itanium® processor. Some of the original assumptions such as template guided instruction issue mechanism are not applicable to dynamic implementation techniques for future Itanium® processors, such as use of out-of-order execution core. However, the restrictive instruction encoding does have consequence on impacting code density. In particular, NOPs have to be introduced into the instruction flow when no suitable instructions of a particular resource can be found to fill a bundle. Once fetched into the pipeline, these NOPs can occupy potentially performance critical resources such as expansion queue slots, which could be better used for instructions that perform useful work.

On Itanium® architecture, instruction execution is dynamically predicated. For an instruction on the correct path, if the predicate is false, the instruction will not result in state write-back therefore its effect is equivalent to that of a NOP. As a result, the predicated false instructions are also wasteful and performance degrading since they not only occupy front-end resources such as expansion queues and issue slots but also resource at the end of the pipeline such as execution units and retirement detection unit, which could otherwise be used by predicated true instructions.

Using the mesocode free-form encoding for the microcode, there are no restrictions on how microcode must be ordered. Therefore it is straightforward to remove NOPs from the code stream. Removing predicated false instructions is more difficult because this directly implies a form of predicate prediction. If the prediction is wrong, there needs to be a recovery mechanism in place to execute the original instruction sequence. As a result, for this study we only examine the effects of NOP and misalignment removal.

3.1.3 Code Density Profile

Figure 2 characterizes the fetch bandwidth inefficiency of Itanium® processors for the selected set of benchmarks. It depicts a dissection of all bundles fetched on the correct path on a machine that can fetch 2-bundle per cache line per cycle, such as the fetch engine implemented in [19]. Here we assume perfect branch prediction.

Starting from the bottom, the first two categories account for the percentages of NOPs and predicated false instructions, respectively. The next two categories represent misalignment. The third category accounts for those fetches due to misalignment at branch targets, corresponding to a fragmentation at the beginning of a cache line, while the fourth category due to taken branches in the middle of a cache line and leave the rest of the line unused, thus representing fragmentation at the end of the cache line. The topmost category represents the truly useful instructions that are fetched and eventually retired.

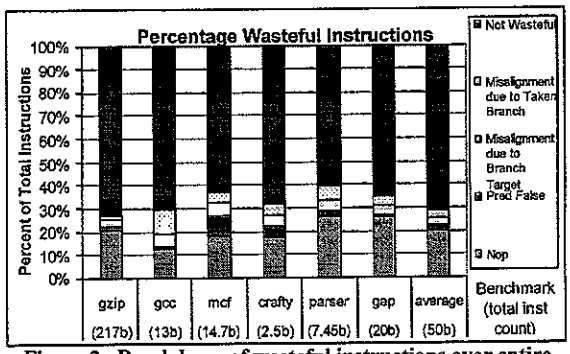

Figure 2. Breakdown of wasteful instructions over entire program execution

On average, nearly 30% of all fetched instructions are wasteful, where NOP accounts for a majority, misalignment second, predicated false trailing as the third. Since branch misprediction has been factored out, the waste in fetch bandwidth is direct manifestation of code density problem. This imposes an ultimate limit on the achievable IPC regardless implementation specifics of any particular processor. Given a good method of identifying candidate regions to encode using mesocode, it should be possible to come close to removing all of the wasteful instructions from the dynamic code stream.

3.2. Stream-based Mesocode

A *stream* is a sequence of instructions between two retired taken branches [24]. Streams can be viewed as coarser-grained primitive units of program execution than basic blocks, making them natural candidates for mesocode encoding. Streams leverage the ideas that many branches are biased and highly predictable, and the strongly biased direction can be made the not taken direction. During program execution the branches within a stream will likely turn out to be not taken, and complete streams will likely be repeatedly executed. Streams can be constructed at compile time using profiling statistics.

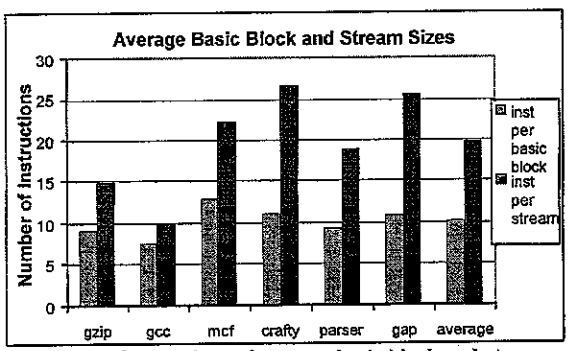

Figure 3. Comparison of average basic block and stream sizes

Figure 3 illustrates the average basic block (inst per basic block) and stream (inst per stream) sizes for our benchmarks. On average, streams are about twice the size of basic blocks. As can be seen, Itanium® basic blocks are relatively large (on average 10 instructions per block) due to the use of predication in removing some of the branches. Operating on streams that average 20 instructions each is a real advantage in instruction fetching, i.e. the sequential control flow is only interrupted once in every 20 instructions. This phenomenon is primarily because the original binaries are produced using an advanced compiler [10] that is capable of sophisticated edge profiling and path profiling that are geared towards optimizing the frequently traversed paths in hot code. In addition, application of predication also helps further remove certain branches from the code. This is the fundamental reason why optimized code such as those produced for Itanium® architecture exhibit dominant existence of streams.

For the purposes of this study, streams provide an easy way of identifying candidate regions of code for mesocode encoding. Streams are better candidates than basic blocks because their larger size cover more dynamic instructions and provide a larger window for optimizations. The resulting stream-based mesocode has the same characteristics expected of streams.

3.2.1. Stream Coverage and Locality

It is possible to characterize an entire program in terms of basic blocks; the same can be done using streams. A program execution can be broken down into a sequence of streams by segmenting instructions between dynamically taken branches. This trace of execution will contain recurring streams and patterns of streams corresponding to loops and phases of program execution.

Figure 4 illustrates the coverage of the overall dynamic instruction count by the dominant streams. In general, about 10% of the streams cover over 90% of the dynamic instructions, which reflects strong locality of streams and indicates the presence of hotspots in the program. Therefore optimizing a few streams can improve the overall performance. Mesocode is primarily targeted on those 10% highest dynamic coverage dominant streams.

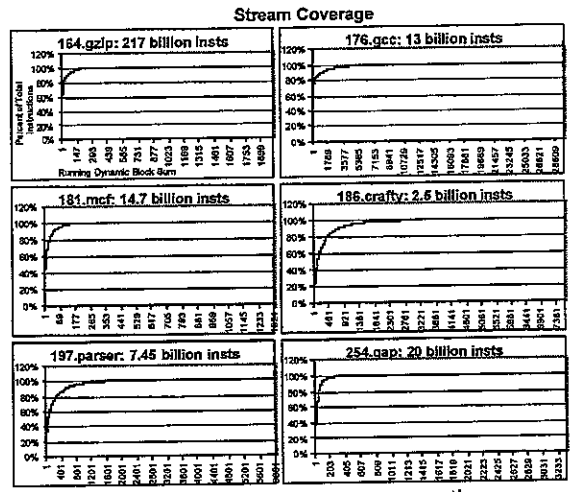

Figure 4. Cumulative stream coverage over entire program execution

To shed light on the potential for code density improvement, Figure 5 characterizes the entire program execution by breaking down the types of dynamic instructions covered and not covered by the dominant streams. For each benchmark, the top five categories represent the percentage of dynamic instructions that are not covered by the dominant streams and would not benefit from stream optimization. On average, these represent less than 10% of all dynamic instructions. The bottom five categories correspond to the instructions that are covered by dominant streams and would benefit from stream-based optimization. On average, about 20% of the dynamic instructions can be removed for code density optimizations. The reductions correspond to removal of NOPs and misaligned branch targets, which are the fourth and second categories from the bottom, respectively.

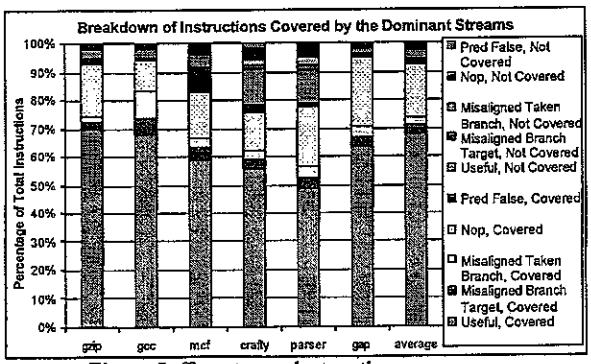

Figure 5. Top stream instruction coverage

In targeting only 10% of the highest dynamic coverage streams, one would expect the static size of these streams to be relatively small when compared with the original code.

Table 1. Static code section sizes

Benchmark	Original Code Instructions	Mesocode Instructions	Dynamic Instruction Coverage
164.gzip	864,900,587	8,712	99.34%
176.gcc	18,819,624,464	123,258	96.63%
181.mcf	563,742,820	4,020	97.89%
186.crafty	3,819,655,366	35,100	91.57%
197.parser	2,692,201,257	21,318	91.39%
254.gap	1,806,018,040	12,738	99.32%

Table 1 illustrates the static number of instructions covered by the top ten percent highest coverage streams as compared to the number of instructions in the original code. The mesocoded streams are cache aligned but have not been compressed or rescheduled. Therefore the number of instructions reported for the mesocode section includes both instructions taken from the streams in the original code and the fragmentation at the end of the cache line due to taken branches. The size of the mesocoded section is much smaller than the original section and represents a very small static code increase. So due to their high dynamic instruction coverage and very small static size, streams make good candidates for mesocode encoding.

Streams are encoded separately from each other in the mesocode section. This is necessary because the intent of mesocode is to express information about groups of instructions, not just the instructions themselves and their optimizations are entirely local to the stream. This information about aggregates of instructions is further useful for extracting interesting information about program behavior.

3.3. Gshare Mesocode Stream Predictor

Mesocode enables one to treat regions of microcode as aggregate units of execution and abstract away from the individual instructions. Treating program execution as a series of mesocoded sections instead of individual instructions or bundles exposes new ways of manipulating program behavior at different levels of granularity. In particular, accurately predicting the next stream to be executed has several potential benefits. First, predicting the next stream is equivalent to predicting the control behavior of a group of instructions, which reduces the need for branch prediction. The effect is to predict multiple branches at once. Second, if predicated false instructions are removed from the streams, stream prediction also acts as predicate prediction for the removed instructions. Third, stream prediction predicts larger groups of instructions than branches earlier in the execution stream, enabling more aggressive instruction prefetch. Since each stream is treated as an aggregate unit of instructions, it is possible to generate some form of instruction prediction upon decoding the escape template at the start of the stream. Therefore generating a prediction using the first instruction of the stream effectively predicts not the next instruction as in the case with branch prediction, but one much farther away in the future. This gives ample time to initiate prefetching of the next stream, which can be triggered by a strategically placed "prepare to branch" instruction in the current stream.

Although a stream predictor could predict the behavior of some regions of code more efficiently, it does not necessarily supplant existing hardware structures such as the branch predictor. The stream predictor could also be used to predict transitions between original and mesocode, so that the appropriate hardware structures in the fetch and decode stages can be bypassed. Table 2 illustrates the four types of control flow transitions, namely, between the original binary, between mesocode (stream), and between stream and original code.

Table 2. Code transition scenarios

Transition Scenarios	Predictor Coverage
Original code -> original code (branches)	Traditional branch predictor
Original -> mesocode (mesocode entry)	Traditional branch predictor
Mesocode -> mesocode	Mesocode predictor
Mesocode -> original code (mesocode exit)	Mesocode predictor

The mesocode predictor implemented for this study uses an adaptation of the gshare [11] branch prediction algorithm because of its relative simplicity. It is possible to adapt any branch prediction algorithm for this purpose.

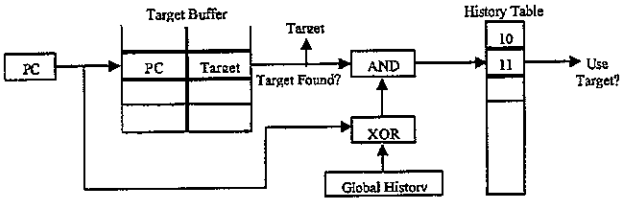

Figure 6. Gshare Predictor

Figure 6 illustrates the structures used in gshare prediction. There are separate tables for storing the PC mappings and the histories for each table. The instruction PC is used to index into the target buffer, which maps a target instruction PC to the given instruction. For a branch predictor, the target PC is the branch target. For the stream predictor, the target PC is the first instruction of the predicted stream. The instruction PC and a global history are then used to index the separate history table. The history table contains two-bit histories that capture the past history of the instruction and generate a prediction. For a branch predictor, the two-bit history predicts whether the branch is taken or not. For the stream predictor, the two-bit history predicts whether the target PC is used or program execution should fall back to the original code.

4. PERFORMANCE SIMULATION METHODOLOGY

To evaluate performance benefit and tradeoffs for mesocode, we carried out experiments by using SMTSIM/IPFsim, a version of SMTSIM simulator [17] adapted to work with Intel Itanium®-based simulation environment [20]. This infrastructure is execution-driven and cycle-accurate. It supports a variety of Itanium® experimental processor models for research purposes, including in-order and out-of-order pipelines.

In our study, the baseline in-order model is a two-bundle wide, 12-stage Itanium® microarchitecture similar to [19], but with longer pipeline to account for higher core pipeline frequency. Compared to the baseline in-order model, the OOO model assumes two additional front-end pipe stages to account for the extra OOO design complexity, and a register rename stage and a scheduler stage, which replaces the expansion queue stage in the baseline. More details on the model are described in Table 3. The pipeline organization is depicted in Figure 7.

Table 3. Details of the modeled research Itanium® processor	
Pipeline Structure	Out-of-order: 16 stage pipeline.
Fetch	2 bundles
Branch Predictor	2K entry GSHARE 256 entry 4-way associative BTB
Stream Predictor	2K entry GSHARE 32 entry 4-way associative BTB
Expansion Queue	In-order 8 bundle queue
Register File	128 Integer Registers, 128 FP Registers, 64 Predicate Registers, 128 Control Registers
Execute Bandwidth	Out-of-order: 18 instruction schedule window
Cache Structure	L1 (separate I and D): 16K 4-way, 8 way banked, 2 cycle latency L2 (shared): 256K 4-way, 8 way banked, 14 cycle latency L3 (shared): 3072K 12-way, 1 way banked, 30 cycle latency Fill buffer (MSHR): 16 entries. All caches have 64 byte lines
Memory Latency	230 cycle latency, TLB Miss Penalty 30 cycles

Benchmarks include six integer benchmarks selected from the SPEC2000int suite. The benchmarks and simulation setup are summarized in Table 4. All benchmarks are simulated for 200 million retired instructions after fast-forwarding past initialization code (with cache warmup). In our initial simulation experiments, much longer runs of the benchmarks were performed, however it was observed that the longer-running simulation results yielded only negligible performance differences.

Table 4. Workload setup			
Benchmark	Input	Fast-forward Distance (30% of workload)	Total Instructions
164.gzip	Lite	18 billion insts	217 billion
176.gcc	Lite	6.6 billion insts	13 billion
181.mcf	Lite	8.8 billion insts	14.7 billion
186.crafty	Lite	1.6 billion insts	2.5 billion
197.parser	Lite	1 billion insts	7.45 billion
254.gap	Lite	10 billion insts	20 billion

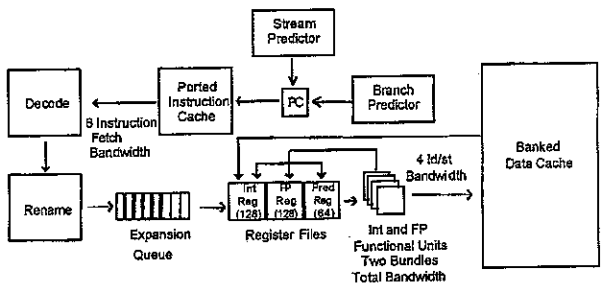

Figure 7. Pipeline organization of a research Itanium® processor

All binaries used in this work are compiled with the Intel Electron compiler for the Itanium® architecture [2, 10], which incorporates the state-of-the-art instruction level parallelism optimization techniques. All benchmarks are compiled with maximum compiler optimizations enabled, including profile-driven feedback-guided optimization, software prefetching, software pipelining, control speculation and data speculation.

5. PERFORMANCE EVALUATION

In this section, a detailed performance evaluation is presented to highlight performance and tradeoffs of mesocode for out-of-order processor pipelines. For out of order processors, the front-end fetch bandwidth is critical to performance. By using mesocode, performance improvement of 32% can be achieved

due to both instruction cache prefetch and the reduction of resource contention as a result of NOP removal from the streams.

A variety of predictor configurations are evaluated, including using branch predictor alone, stream predictor alone and a hybrid of both. The baseline reference model is an out of order processor that uses traditional gshare branch predictor alone with a 4-way set associative, 256-entry BTB and a 2048-entry (12-bit global history) history table of 2-bit counters. For the base model of stream predictor, the BTB is 4-way set associative with 32 entries, and the history table continues to have 2048 entries.

5.1. Out-of-order Performance

Unlike in-order processor, out-of-order execution allows the processor to dynamically schedule the instruction for execution therefore its performance is much more sensitive to availability of critical pipeline resource including fetch bandwidth and core resources like various instructions queues. In general, out-of-order core demands more instructions to be fetched into the pipeline in order to overlap their execution with concurrent servicing of long latency operations such as outstanding cache misses or multi-cycle function executions. Any contention and effective misuse by wasteful instructions such as NOPs on these critical pipeline resources can lead to severe starvation in the aggressive out-of-order execution engine. Thus, fetch bandwidth becomes a critical performance-limiting factor for the out-of-order processor. On the contrary, an in-order processor by definition must execute its instructions sequentially, following compiler's static scheduling. The pipeline stalls either due to encountering stop bit inserted by the compiler or upon an instruction using results of an outstanding cache miss. Thus, the performance bottleneck of the in-order processor is usually not the fetch bandwidth or contention at front-end resources like expansion queues.

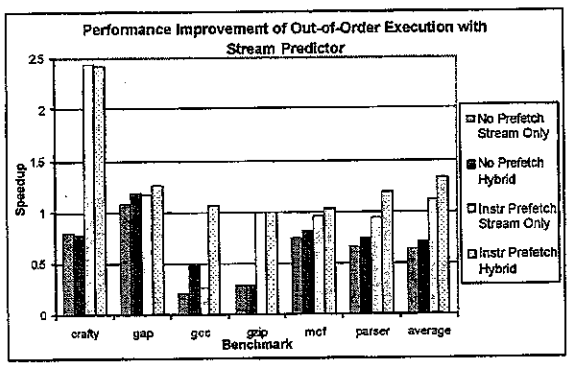

Figure 8. Performance of out-of-order execution with stream predictor

The third and fourth bars in Figure 8 illustrate the same set of predictor configurations, but using "prepare to branch" in streams to enable instruction prefetching. In addition, NOPs are removed from stream, thus representing further optimization in mesocode. Note that instruction prefetching is only used for the stream code. Since more instructions can be fetched before the control flow changes, the use of instruction prefetching is exceptionally helpful for the stream code.

The baseline is the out-of-order model described in Section 4 and all performance are normalized to that of the baseline out-of-order model. For the out-of-order processor with instruction prefetching and the hybrid predictor, we see a 32% performance improvement over the baseline, much better than the improvement achieved on the in-order processor. Even for the configuration that only the stream predictor without branch predictor, but with instruction prefetching, (third bar from the left), the performance is 13% higher than the baseline. For crafty, the improvement is over 140%, which clearly indicates that this benchmark is limited by fetch bandwidth. In addition, improving fetch bandwidth via reduction of NOPs from the streams also greatly help lessen resource contention on critical resources like ROB and instructions queues in the pipeline.

The performance degradation in the cases without prefetching indicate that despite the code density improvements from NOP removal and realignment, instruction cache penalties still take a toll on performance. Gap shows improvement even without prefetching because it has far fewer taken branches, with roughly seven fall-through branches for every taken branch. As a result, when branch prediction is taken away, gap still does not suffer very much from taken branch penalties and this is enough to compensate any degradation due to instruction cache misses.

The high instruction cache penalty is likely due to conflict misses stemming from aliasing in the addresses used by the mesocode section. As a result, there is a high miss stall rate. Performance improves overall if prefetching and code density optimizations are combined. This is because prefetching can be initiated early enough to overcome the effects of conflict misses, while the code density optimizations provide the pipeline with enough instructions to issue every cycle.

In most cases, performance is higher with the hybrid predictor configuration than with the mesocode predictor alone. This is because all branches in the non-mesocode code are effectively predicted not taken, and there are enough taken branches to cause substantial penalties. Only crafty and gzip show a very slight degradation when going to the hybrid predictor, which indicates that few such branches exist. This data also indicates that potentially it is likely better to keep the branch and stream predictors and their histories separate from each other as the behavior of their predictions are potentially quite different and have little to do with each other. They are accessed at different times: the stream predictor in the decode stage or later (after the escape template is encountered to indicate demarcation of a stream), and the branch predictor in the fetch stage. The former is more latency tolerant and can be performed off the critical path and could be implemented in more than 1 cycle, while the latter is more timing critical and usually is implemented in 1 cycle. Since both structures would be smaller than a traditional branch predictor, decoupling their behavior also makes their layout and placement more flexible, especially, the stream predictor can be physically moved away from the critical path. In addition, due to much lessened activities on the stream predictor compared to the activities on the normal branch predictor, specific design of stream predictor may also optimize

for lower power concerns. The benefits and tradeoffs are beyond the scope of this paper.

6. RELATED WORK

Extensive research works have been conducted in the area of continuously improving effective fetch bandwidth. Traditional hardware approaches such as trace cache [16] and replay [5] and block-level ISA [3] mechanisms employ specialized hardware to dynamically capture frequently executed instruction sequence, represent them into internal format and store in microarchitectural structure such as trace cache or frame caches from which most instructions are expected to be supplied.

Software approaches such as software trace cache [15], Dynamo [1] and Spike [25] are geared towards achieving similar benefit of trace cache by identifying the frequently execution path in the program either dynamically or through profile, then reorder basic blocks to bias towards frequent non-taken cases. This enables long sequences instructions are frequently fetched without being interrupted by taken branches.

In perspective, mesocode literally is a mechanism between pure hardware approach like trace cache or replay and the software approaches such as the software trace cache. It is a hybrid software and hardware approach. The key motivation behind mesocode is to make small changes to the hardware and software so as to make the overall front end more efficient. In particular, for EPIC ISA like Itanium® ISA, mesocode provides a means to unshackle the oftentimes inflexible and non-scalable instruction bundling restriction for the frequently executed portion of the program. This relaxation can enable post-pass software binary adaptation of legacy binary built for canonical Itanium® architecture to work more efficiently on future processor implementation, which may be much more aggressive than a canonical in-order implementation.

7. CONCLUSIONS

The key insight is that for out-of-order processor, the front-end fetch bandwidth is a critical resource. By using mesocode, performance improvement of 32% can be achieved due to both instruction cache prefetch and reduction of resource contention as result of NOP removal from the streams. Many benchmarks suffer substantial branch misprediction penalties in their original code if those branches are always predicted not taken. Furthermore, streams tend to alias in the instruction cache which causes miss penalties on instruction fills. The combination of these two degrading factors is enough to void the gains from NOP removal alone. As a result, NOP removal must be used in conjunction with instruction prefetch.

In this work, we only examine the application of mesocode in uniprocessor single threaded application, as multithreaded hardware techniques are adopted in main stream microprocessor designs [26, 27], and resource contention at front-end will become even more crucial, it would be interesting to examine how to mesocode can be applied to achieve performance improvement.

8. REFERENCES

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *Proc. ACM SIGPLAN '00 Conf. On Programming Language Design and Implementation*, pp. 1-12, 2000.

[2] J. Bharadwaj et al. The Intel IA-64 Compiler Code Generator, *IEEE Micro*, Sept-Oct 2000, pp. 44-53.

[3] P.-Y. Chang, E. Hao, and Y. N. Patt. Target Prediction for Indirect Jumps. In *Proc. 24th Int'l Symp. on Computer Architecture*, pp. 274-283, June 1997.

[4] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. In *Proc. 28th Int'l Symp. on Computer Architecture*, pp. 14-25, July 2001.

[5] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S. J. Patel, and S. S. Lumetta. Performance Characterization of a Hardware Framework for Dynamic Optimization. In *Proc. 34th Int'l Symp. on Microarchitecture*, December 2001.

[6] B. Fields, S. Rubin, and R. Bodik. Focusing Processor Policies via Critical-Path Prediction. In *Proc. 28th Int'l Symp. on Computer Architecture*, pp. 74-85, July 2001.

[7] E. Hao, P. Chang, M. Evers, and Y. N. Patt. Increasing the Instruction Fetch Rate via Block-Structured Instruction Set Architectures. In *Proc. 29th Int'l Symp. on Microarchitecture*, pp. 191-200, December 1996.

[8] D. Joseph and D. Grunwald. Prefetching Using Markov Predictors. In *Proc. 24th Int'l Symp. on Computer Architecture*, pp. 252-263, 1997.

[9] S. Jourdan, T. H. Hsing, J. Stark, and Y. N. Patt. The Effects of Mispredicted-Path Execution on Branch Prediction Structures. In *Proc. 1996 Conf. On Parallel Architectures and Compilation Techniques*, October 21-23, 1996.

[10] R. Krishnaiyer, D. Kulkarani, D. Lavery, W. Li, C. Lim, J. Ng and D. Sehr, An Advanced Optimizer for the IA-64 Architecture, *IEEE Micro*, Nov-Dec 2000.

[11] S. McFarling Combining Branch Predictors. Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[12] M. C. Merten, A. Trick, E. Nystrom, R. Barnes, and W. Hwu. A Hardware Mechanism for Dynamic Extraction and Relayout of Program Hot Spots. In *Proc. 27th Int'l Symp. on Computer Architecture*, pp. 59-70, June 2000.

[13] K. Pettis and R. C. Hansen. Profile Guided Code Positioning. In *Proc. ACM SIGPLAN 1990 Conf. On Programming Language Design and Implementation*, pp. 16-27, June 1990.

[14] E. M. Nystrom, R. Barnes, M. Merten, and W. Hwu. Code Reordering and Speculation Support for Dynamic Optimization Systems. In *Proc. Int'l Conf. On Parallel Architectures and Compilation Techniques*, September 2001.

[15] A. Ramirez, J. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero. Software Trace Cache. In *Proc. 1999 Int'l Conf. On Supercomputing*, June 1999.

[16] E. Rotenberg, S. Bennett, and J. E. Smith. Trace Cache: a low latency approach to high bandwidth instruction fetching. In *Proc. 29th Int'l Symp. on Microarchitecture*, pp. 24-34, December 1996.

[17] D. M. Tullsen. Simulation and modeling of a simultaneous multithreaded processor. In *22nd Annual Computer Measurement Group Conference*, December 1996.

[18] E. Tune, D. Liang, D. M. Tullsen and Brad Calder. Dynamic Prediction of Critical Path Instructions. In *Proc. 7th Int'l Symp. on High Performance Computer Architecture*, January 2001.

[19] H. Sharangpani and K. Aurora, Itanium® Processor Microarchitecture. *IEEE Micro*, Sept.-Oct. 2000, pp. 24-43.

[20] R. Uhlig, R. Fishtein, O. Gershon, I. Hirsh, and H. Wang. SoftSDV: A Presilicon Software Development Environment for the IA-64 Architecture. In *Intel Technology Journal*, Q4 1999.

[21] C. Zilles and G. Sohi. Execution-based Prediction Using Speculative Slices. In *Proc. 28th Int'l Symp. on Computer Architecture*, July 2001.

[22] Intel Corp. Intel IA-64 Architecture Software Developer's Manual.

[23] SPEC. SPEC CPU2000 Documentation (www.spec.org/osg/cpu2000/docs/)

[24] A. Ramirez, J. Larriba-Pey, and M. Valero. A Stream Processor Front-end. In IEEE TCCA Newsletter, June 2000.

[25] A. Ramirez, L. A. Barroso, K. Gharachorloo, G. Lawney, R. Cohn, J. L. Larriba-Pey and M. Valero. Code Layout Optimizations for Transaction Processing Workloads. In *Proc. 28th Int'l Symp. on Computer 'Architecture*, July 2001.

[26] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, Q1 2001.

[27] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *22nd International Symposium on Computer Architecture*, June 1995.